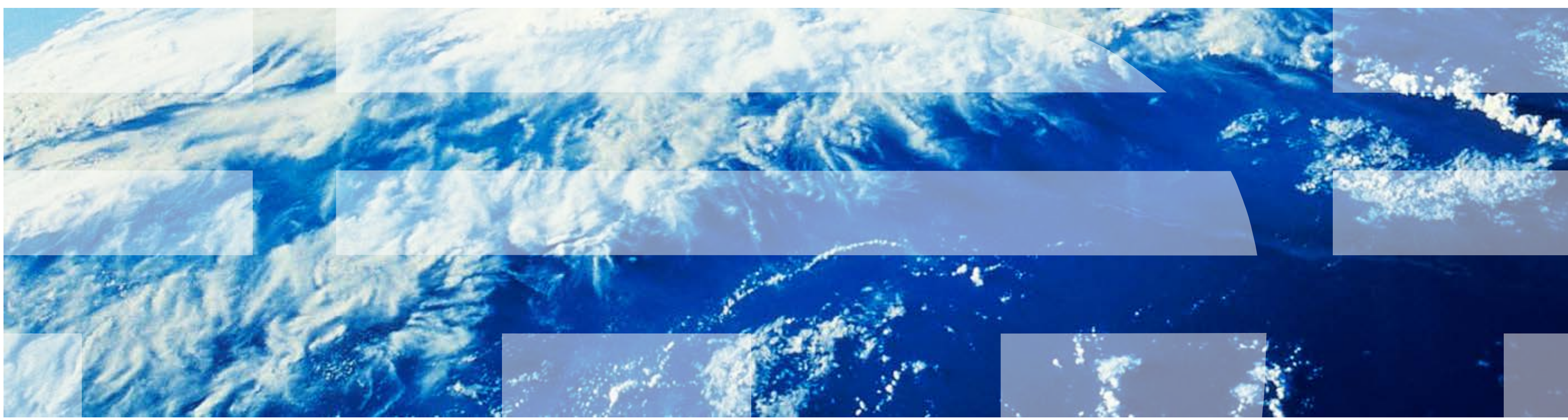
Borrowed from Shiva Shivakumar

Computer Systems for Data Science
Topic 2

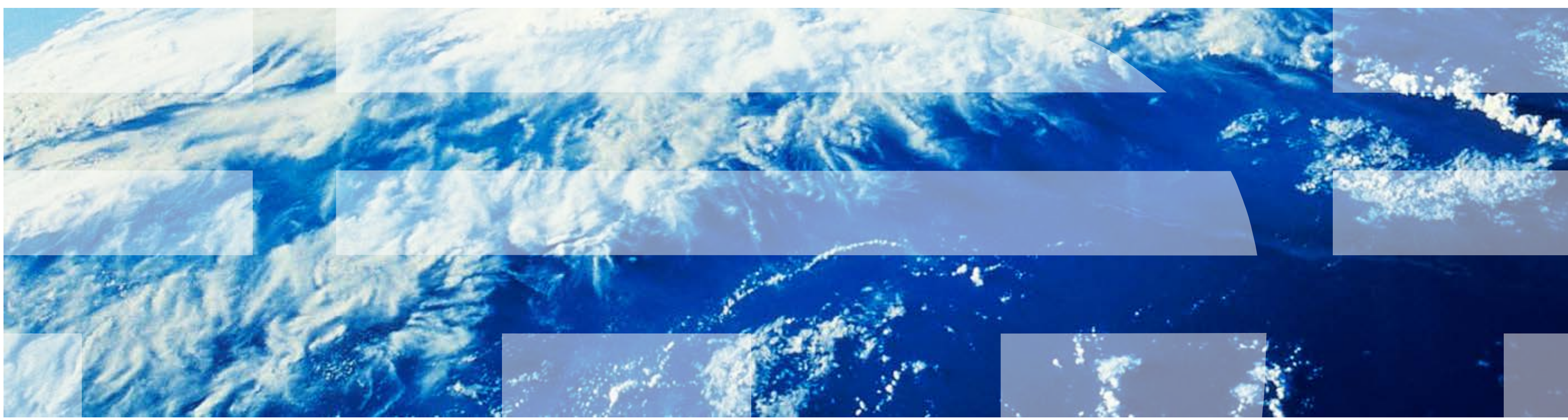
Relational Model and SQL



What we'll cover in this topic

- **Intuition**
  - Basic relational model
  - Map-filter-reduce concept
- **Intro to SQL**
  - Schemas, query structure of SELECT-FROM-WHERE, JOINs

Relational Model: Intution

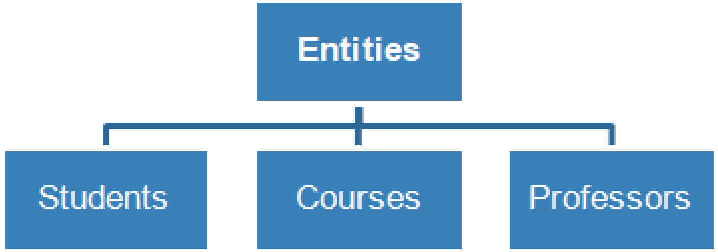


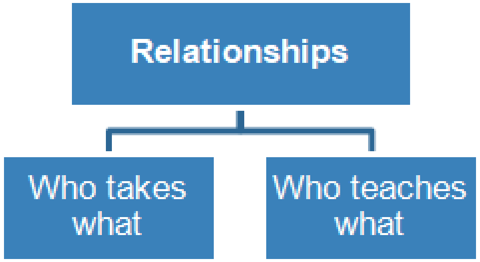
A Motivating Example

A basic Course Management System (CMS):

Entities (e.g., Students, Courses)

Relationships (e.g., Alice is enrolled in CSEE 4121)





Intuition: Spreadsheet Tables

Logical Schema

Student(cuid: string, name: string, gpa: float)

Courses(cid: string, c-name: string, room: string)

Enrolled(cuid: string, cid: string, grade: string)

Students

CUID	Name	GPA
as2121	Alice Smith	4.3
jg9999	Jay Goodwin	1.2
mc2312	Min Chang	2.2
zb1111	Zorn Bjorn	3.8

Enrolled

CUID	CID	Grade
as2121	4121	A+
jg9999	4121	C
mc2312	3292	A+
zb1111	2999	D

Courses

CID	C-Name	Room
4121	Computer Sy	CEPSR
3292	Databases	MUDD 1
2999	Algorithms	MUDD 2

<u>Queries [“compute” over tables]</u>

- Alice’s GPA?
- Jay’s classes?
- AVG student GPA?
- AVG student GPA in CSEE 4121?

Relational Model and Schemas

- Definition: Data model
  - Organizing principle of data + operations
- Relational model (aka tables)
  - Simple and most popular
  - Elegant algebra (E.F. Codd et al)
- Definition: Schema
  - Describes blueprint of table (s)
- Every relation has a schema
  - Logical Schema: describes types, names
  - Physical Schema: describes data layout
  - Virtual Schema (Views): derived tables

Data Independence

1. Can we add a new column or attribute without rewriting the application?

   – **Logical Data Independence**
     - Protection from changes in the logical structure of the data

2. Do you need to care which disks/machines are the data stored on?

   – **Physical Data Independence**
     - Protection from Physical Layout Changes

Python Operating on Lists

Basic types

- Int
- Long
- String

Map + Filter

- map(function, list)
- filter(function, list)

Map applies function to input list
Filter returns sub list that satisfies filter condition

Reduce/Aggregate

- reduce(…)

Reduce runs a computation on a list and returns a result
E.g., SUM, AVG, MAX

SQL Queries on Tables (Lists of Rows)

Basic types

- Int32, Int64
- Char[n]
- Float32, Float64

Map + Filter

Single Table Query

```
SELECT c1, c2
FROM   T
WHERE condition;
```

Multi Table JOIN

```
SELECT c1, c2
FROM   T1, T2
WHERE condition;
```

Reduce/Aggregate

```
SELECT SUM(c1*c2)
FROM   T
WHERE condition
GROUP BY c3;
```

Map-Filter-Reduce pattern: Same simple/powerful idea in MapReduce, Hadoop, Spark, etc.

SQL Cheat Sheet (www.sqltutorial.org/sql-cheat-sheet)

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

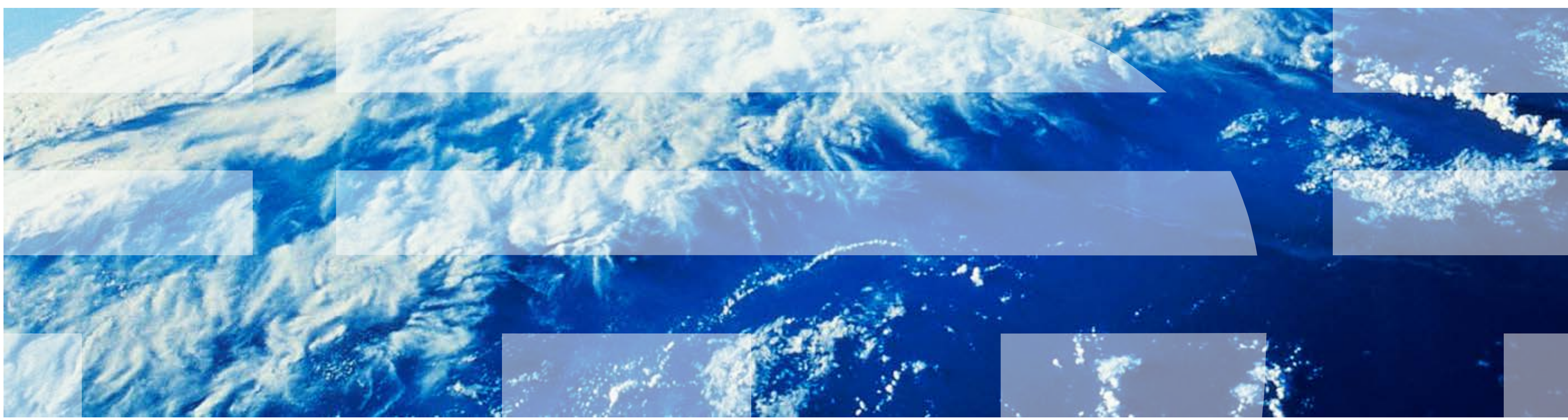
```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

Intro to SQL



SQL Introduction

- SQL is a standard language for querying and manipulating data

- SQL is a **very high-level** programming language
  - This works because it is optimized well!

SQL stands for
Structured
Query
Language

SQL is a…

- Data Manipulation Language (DML)
    - Query one or more tables
        - Insert/delete/modify tuples in tables

- Data Definition Language (DDL)
    - Define relational schemata
    - Create/alter/delete tables and their attributes

Basic Set Algebra Concepts

- List: [1, 1, 2, 3] Ordered, duplicates
- Set: {2, 1, 3} Unordered, no duplicates
- Multiset: {2, 1, 3, 1} Unordered, duplicates

- Unions:
  - Set: {2, 1, 3} U {2, 3} = {2, 1, 3}
  - Multiset: {2, 1, 3} U {2, 3} = {2, 1, 3}

- Cross-product:
  - {1, 1, 2, 3} * {y, z} = {1, y}, {1, y}, {2, y}, {3, y}, {1, z}, {1, z}, {2, z}, {3, z}

Tables in SQL

Product

PName	Price	Manuf
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

A **relation** or **table** is a multiset of tuples/rows having the attributes specified by the schema

Tables in SQL

Product

PName	Price	Manuf
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

An **attribute** (or **column**) is a typed data entry present in each tuple in the relation

*NB: Attributes must have an **atomic** type in standard SQL, i.e. not a list, set, etc.*

Tables in SQL

Product

PName	Price	Manuf
Gizmo	**$19.99**	**GizmoWorks**
Powergizmo	**$29.99**	**GizmoWorks**
SingleTouch	**$149.99**	**Canon**
MultiTouch	**$203.99**	**Hitachi**

A **tuple** or **row** or **record** is a single entry in the table having the attributes specified by the schema

Tables in SQL

Product

PName	Price	Manuf
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

The number of tuples is the **cardinality** of the relation

The number of attributes is the **arity** of the relation

Data Types in SQL

Atomic types:

- Characters: CHAR(20), VARCHAR(50)
- Numbers: INT, BIGINT, SMALLINT, FLOAT
- Others: MONEY, DATETIME…

Every attribute must have an atomic type

Table Schemas

The **schema** of a table is the table name, its attributes, and their types:

Product(Pname: *string*, Price: *float*, Category: *string*, Manufacturer: *string*)

A **key** is an attribute whose values are unique; we underline a key

Product(<u>Pname</u>: *string*, Price: *float*, Category: *string*, Manufacturer: *string*)

Key constraints

A **key** is a **minimal subset of attributes** that acts as a unique identifier for tuples in a relation

- A key is an implicit constraint on which tuples can be in the relation
- i.e. if two tuples agree on the values of the key, then they must be the same tuple!

Students(cuid:string, name:string, gpa: float)

1. Which would you select as a key?
2. Is a key always guaranteed to exist?
3. Can we have more than one key?

Declaring Schema

Students(<u>cuid</u>: *string*, name: *string*, gpa: *float*)

```
CREATE TABLE Students (
  cuid CHAR(20),
  name  VARCHAR(50),
  gpa  float,
  PRIMARY KEY (cuid),
)
```

NULL and NOT NULL

- To say "don't know the value" we use NULL

Students(cuid:string, name:string, gpa: float)

cuid	name	gpa
123	Alice	3.9
143	Jim	NULL

Say, Jim just enrolled in his first class.

In SQL, we may constrain a column to be NOT NULL, e.g., "name" in this table

General Constraints

- We can actually specify arbitrary assertions
  E.g. "*There cannot be 25 people in the DB class*"
- In practice, we don't specify many such constraints. Why?
  Performance!

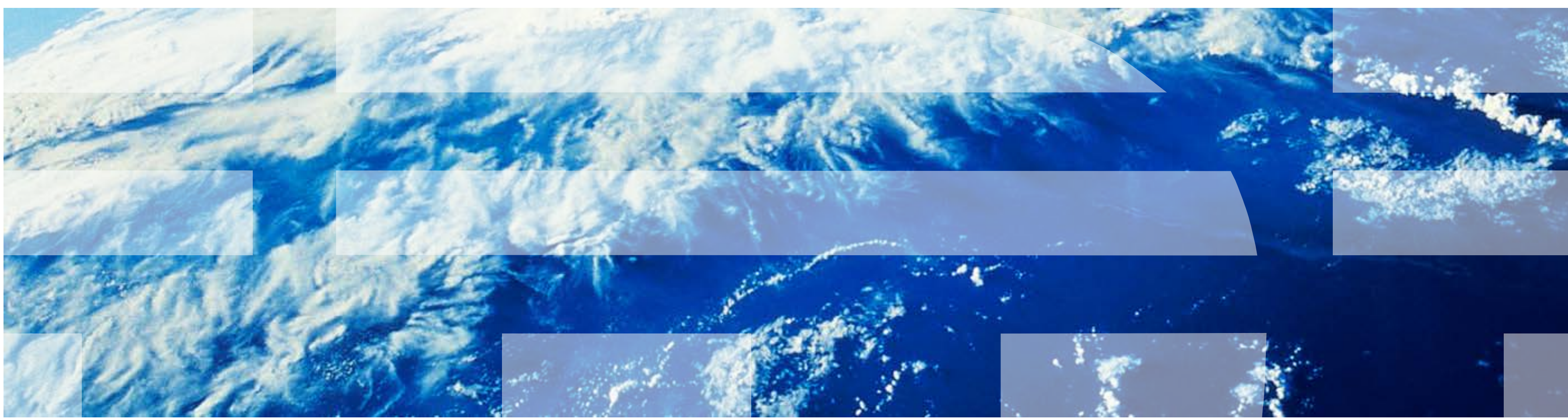
Whenever we do something ugly (or avoid doing something convenient) it's for the sake of performance

Summary of Schema Information

- Schema and Constraints are how databases understand the semantics (meaning) of data
- SQL supports general constraints:

  Keys and foreign keys are most important

Single Table Query



SQL Query

Basic form (there are many many more bells and whistles)

SELECT <attributes>

FROM <one or more relations>

WHERE <conditions>

Call this a **SFW** query.

Simple SQL Query: Selection

Selection is the operation of filtering a relation’s tuples on some condition

```
SELECT *
FROM   Product
WHERE  Category = ‘Gadgets’
```

PName	Price	Category	Manuf
Gizmo	$19.99	Gadgets	GWorks
Powergizmo	$29.99	Gadgets	GWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

PName	Price	Category	Manuf
Gizmo	$19.99	Gadgets	GWorks
Powergizmo	$29.99	Gadgets	GWorks

Simple SQL Query: Projection

Projection is the operation of producing an output table with tuples that have a subset of their prior attributes

SELECT Pname, Price, Manufacturer

FROM Product

WHERE Category = 'Gadgets'

PName	Price	Category	Manuf
Gizmo	$19.99	Gadgets	GWorks
Powergizmo	$29.99	Gadgets	GWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

PName	Price	Manuf
Gizmo	$19.99	GWorks
Powergizmo	$29.99	GWorks

Notation

Input Schema

Product(<u>PName</u>, Price, Category, <u>Manufacturer</u>)

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```

Output Schema

Answer(PName, Price, Manfacturer)

A Few Details

- SQL **commands** are case insensitive:

  Same: SELECT, Select, select

  Same: Product, product

- **Values** are **not:**

  <u>Different:</u> ‘Seattle’, ‘seattle’

- Use single quotes for constants:

  ‘abc’ - yes

  “abc” - no

LIKE: Simple String Pattern Matching

```
SELECT *
FROM   Products
WHERE  PName LIKE '%gizmo%'
```

- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

DISTINCT: Eliminating Duplicates

```
SELECT DISTINCT Category
FROM   Product
```

Category
Gadgets
Photography
Household

Versus

```
SELECT Category
FROM   Product
```

Category
Gadgets
Gadgets
Photography
Household

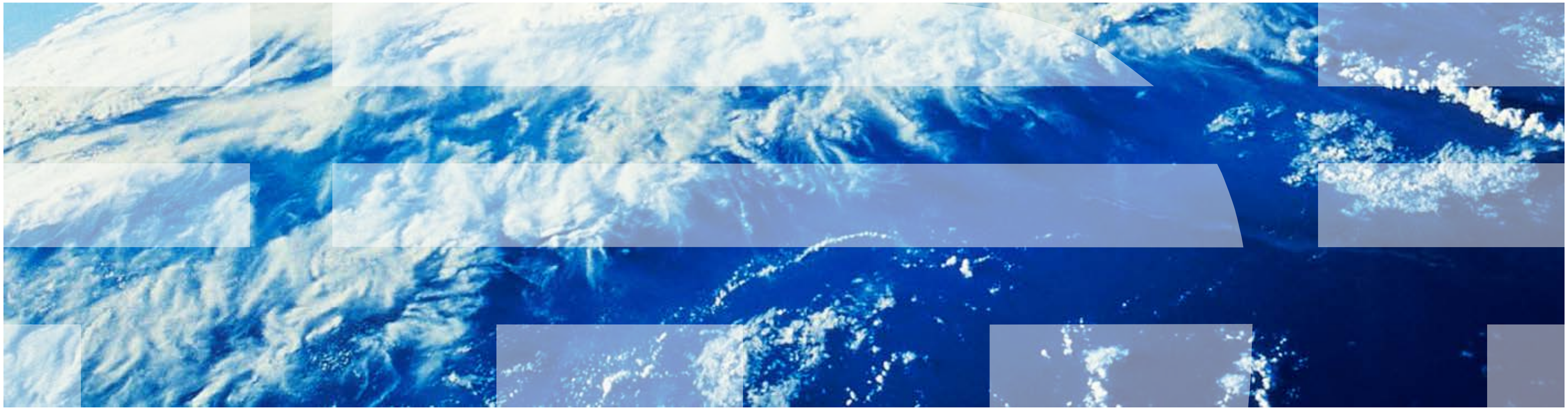
ORDER BY: Sorting the Results

```
SELECT     PName, Price, Manufacturer
FROM       Product
WHERE      Category='gizmo' AND Price > 50
ORDER BY   Price, PName
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

Multi-Table Query



Foreign Key constraints

- Suppose we have the following schema :

> Students(<u>cuid</u>: *string*, name: *string*, gpa: *float*)
>
> Enrolled(<u>student_id</u>: *string*, <u>cid</u>: *string*, grade: *string*)

- And we want to impose the following constraint:

<u>Only bona fide students may enroll in courses'</u> i.e. a student must appear in the Students table to enroll in a class

Students

cuid	name	gpa
102	**Bob**	**3.9**
123	**Mary**	**3.8**

Enrolled

student_id	cid	grade
102	564	A
123	537	A+

We say that cuid is a <u>foreign key</u> that refers to Students

Declaring Foreign Keys

Students(<u>cuid</u>: *string,* name: *string,* gpa: *float*)
Enrolled(student_id: *string*, cid: *string,* grade: *string*)

```
CREATE TABLE Enrolled (
  student_id CHAR(20),
  cid  CHAR(20),
  grade  CHAR(10),
  PRIMARY KEY (student_id, cid),
  FOREIGN KEY (student_id) REFERENCES Students(cuid)
)
```

Foreign Keys and update operations

Students(cuid*: string,* name: *string*, gpa: *float*)

Enrolled(student_id: *string,* cid: *string*, grade: *string*)

- What if we insert a tuple into Enrolled, but no corresponding student?

  INSERT is rejected (foreign keys are constraints)!

DBA chooses

- What if we delete a student?

  1. Disallow the delete
  2. Remove all of the courses for that student
  3. *SQL allows a third via NULL*

Keys and Foreign Keys

Company

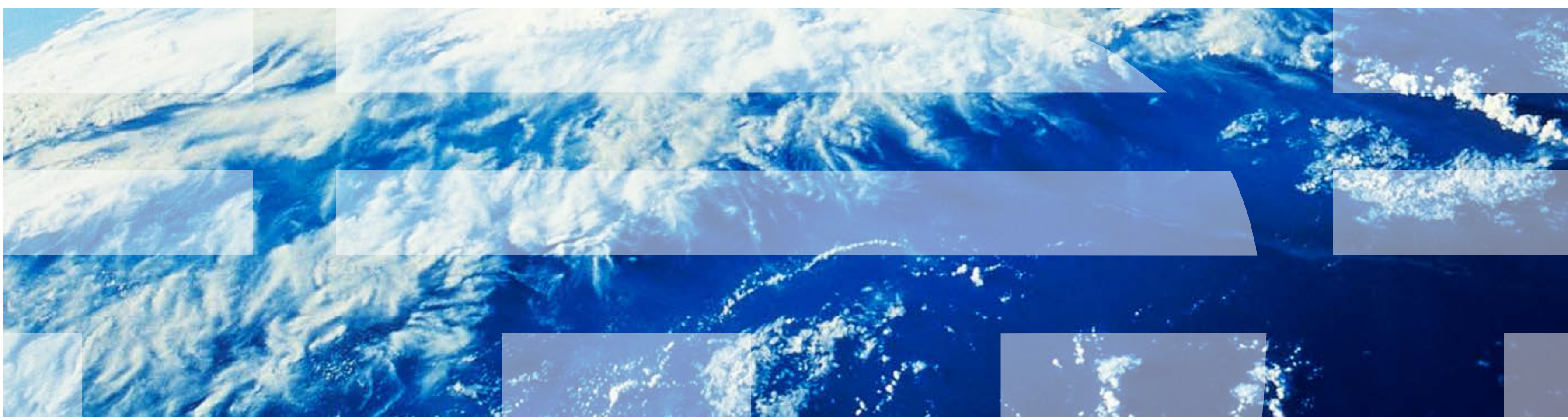
<u>CName</u>	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

What is a foreign key vs. a key here?

Product

<u>PName</u>	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

JOINs and Aggregations



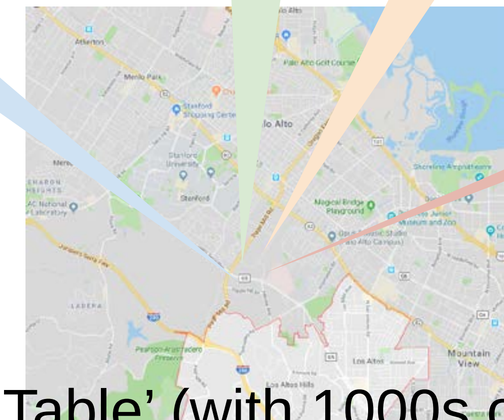
Option 1: Organized tables, with 10s-100s of columns)

SolarPanel			
Zipcode	KW-Total	Carbon offset	...
94305	14.4	29	
94040	32.1	42	
94041	29.1	37.38	

Census	
Zipcode	Population Census
94305	14301
94040	20301
94041	189

Bike share locations		
Zipcode	Lat	Lng
94305	35.1	122.12
94305	35.2	122.13
94041	35.1	121.27
94041		
94041		

Pollution	
Zipcode	Particle count
94305	40
94040	22
94041	57



Option 2: 'FrankenTable' (with 1000s of columns)

Omnidata							
	SolarPanel			Census	Pollution	Bike share locations	
Zipcode	KW-Total	Carbon offset	...	Population	Particle count	Lat	Lng
94305	14.4	29		14301	40	35.1	122.12
94305						35.2	122.13
94305						35.2	122.1
94305						35.1	122.12
94305						...	...

????

Option 1: Organized tables, with 10s-100s of columns)

SolarPanel			
Zipcode	KW-Total	Carbon offset	...
94305	14.4	29	
94040	32.1	42	
94041	29.1	37.38	

Census	
Zipcode	Population Census
94305	14301
94040	20301
94041	189

Bike share locations		
Zipcode	Lat	Lng
94305	35.1	122.12
94305	35.2	122.13
94041	35.1	121.27
94041		
94041		

Pollution	
Zipcode	Particle count
94305	40
94040	22
94041	57

Trade offs?

- Reads? Writes?
- 100s - thousands of applications reading/writing data

⇒ Hybrids: 1 column ⟶ all columns

(Week 7: What's a good schema design?)

Option 2: 'FrankenTable' (with 1000s of columns)

Omnidata							
	SolarPanel			Census	Pollution	Bike share locations	
Zipcode	KW-Total	Carbon offset	...	Population	Particle count	Lat	Lng
94305	14.4	29		14301	40	35.1	122.12
94305	14.4	29		14301	40	35.2	122.13
94305	14.4	29		14301	40	35.2	122.1
94305	14.4	29		14301	40	35.1	122.12

Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Ex: Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
         AND Price <= 200
```

A **join** between tables returns all unique combinations of their tuples **which meet some specified join condition**

Joins

Product

PName	Price	Category	Manufacturer
Gizmo	$19	Gadgets	GizmoWorks
Powergizmo	$29	Gadgets	GizmoWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

CName	Stock Price	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
         AND Country='Japan'
                 AND Price <= 200
```

PName	Price
SingleTouch	$149

An example of SQL semantics

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

Output

A
3
3

R

A
1
3

S

B	C
2	3
3	4
3	5

Cross Product

A	B	C
1	2	3
1	3	4
1	3	5
3	2	3
3	3	4
3	3	5

Apply Selections / Conditions

A	B	C
3	3	4
3	3	5

Apply Projection

Note: we say “semantics” not “execution order”

- The preceding slides show *what a join means*
- Not actually how the DBMS executes it under the covers

Aggregation

```
SELECT AVG(price)
FROM   Product
WHERE  maker = “Toyota”
```

```
SELECT COUNT(*)
FROM   Product
WHERE  year > 1995
```

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

Except COUNT, all aggregations apply to a single attribute

More Examples

Purchase(product, date, price, quantity)

```
SELECT SUM(price * quantity)
FROM   Purchase
```

What do these mean?

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = ‘bagel’
```

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = ‘bagel’
```

⇒ 50 (= 1*20 + 1.50*20)

Grouping and Aggregation

Purchase(product, date, price, quantity)

```
SELECT   product,
             SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Find total sales after 10/1/2005 per product.

Let's see what this means…

Grouping and Aggregation

```
SELECT   product,
              SUM(price * quantity) AS TotalSales
FROM       Purchase
WHERE    date > ‘10/1/2005’
GROUP BY product
```

<u>Semantics of the query:</u>

1. Compute the FROM and WHERE clauses

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: grouped attributes and aggregates

1. Compute the FROM and WHERE clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the GROUP BY

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY

<table>
<tr><th>Product</th><th>Date</th><th>Price</th><th>Quantity</th></tr>
<tr><td rowspan="2">Bagel</td><td>10/21</td><td>1</td><td>20</td></tr>
<tr><td>10/25</td><td>1.50</td><td>20</td></tr>
<tr><td rowspan="2">Banana</td><td>10/3</td><td>0.5</td><td>10</td></tr>
<tr><td>10/10</td><td>1</td><td>10</td></tr>
</table>

3. Compute the SELECT clause: grouped attributes and aggregates

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT →

Product	TotalSales
Bagel	50
Banana	15

HAVING Clause

```
SELECT   product, SUM(price*quantity)
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
HAVING   SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers

HAVING clauses contains conditions on **aggregates**

Whereas WHERE clauses condition on ***individual tuples…***

RECAP: Joins

By default, joins in SQL are **“inner joins”:**

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product
  JOIN Purchase ON Product.name = Purchase.prodName
```

Both equivalent:
Both INNER JOINS!

Outer Joins

- An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations
  - I.e. If we join relations A and B on a.X = b.X, and there is an entry in A with X=5, but none in B with X=5…
  - A LEFT OUTER JOIN will return a tuple (a, NULL)!

- Left outer joins in SQL:

```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase ON
                 Product.name = Purchase.prodName
```

Now we'll get products even if they didn't sell

INNER JOIN

Product

name	category
iphone	media
Tesla	car
Ford Pinto	car

Purchase

prodName	store
iPhone	Apple store
Tesla	car
iPhone	Apple store

```
SELECT Product.name, Purchase.store
FROM   Product
  INNER JOIN Purchase
                      ON Product.name = Purchase.prodName
```

name	store
iPhone	Apple store
iPhone	Apple store
Tesla	car

Note: another equivalent way to write an INNER JOIN!

LEFT OUTER JOIN

Product

name	category
iphone	media
Tesla	car
Ford Pinto	car

Purchase

prodName	store
iPhone	Apple store
Tesla	car
iPhone	Apple store

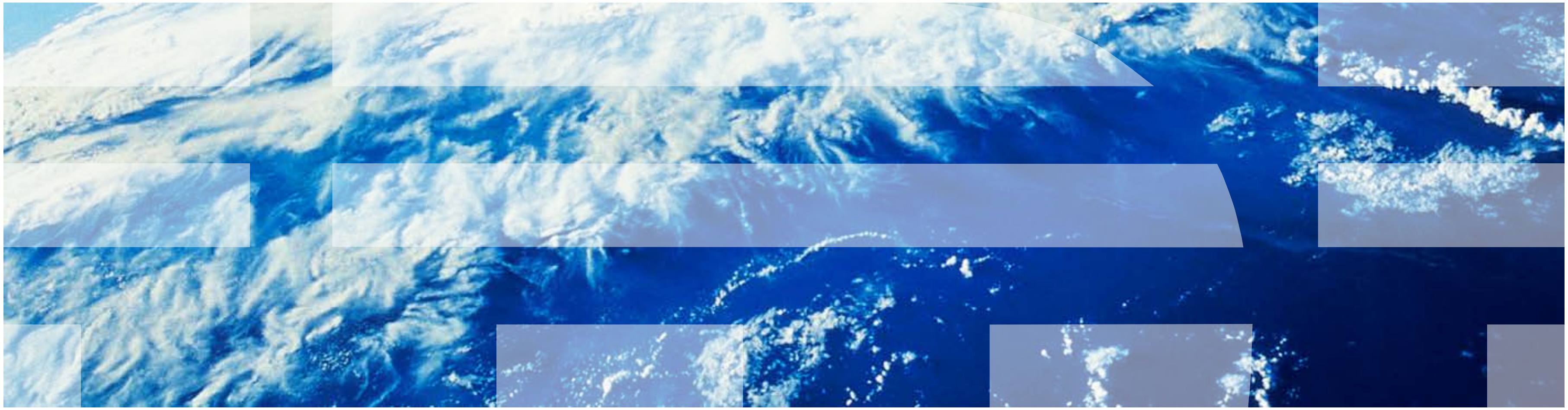
```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase
                        ON Product.name = Purchase.prodName
```

name	store
iPhone	Apple store
iPhone	Apple store
Tesla	car
Ford Pinto	NULL

Other Outer Joins

- Left outer join:
  - Include the left tuple even if there’s no match
- Right outer join:
  - Include the right tuple even if there’s no match
- Full outer join:
  - Include the both left and right tuples even if there’s no match

Nested Queries



SQL is Compositional

Can construct powerful query chains (e.g., f(g(...(x)))

Inputs / outputs are multisets

⇒ Output of one query can be input to another (nesting)!

⇒ Including on same table

Nested queries: Sub-queries Return Relations

Company(<u>name</u>, city)
Product(<u>name</u>, maker)
Purchase(<u>id</u>, product, buyer)

```
SELECT  pr.maker
FROM    Purchase p, Product pr
WHERE   p.product = pr.name
            AND p.buyer = 'Mickey')
```

1. Companies making products bought by Mickey”
2. Location of companies?

Nested queries: Sub-queries Return Relations

Company(<u>name</u>, city)
Product(<u>name</u>, maker)
Purchase(<u>id</u>, product, buyer)

```
SELECT c.city
FROM   Company c
WHERE  c.name  IN (
        SELECT  pr.maker
        FROM    Purchase p, Product pr
        WHERE   p.product = pr.name
                AND p.buyer = ‘Mickey‘)
```

1. Companies making products bought by Mickey”
2. Location of companies?

Subqueries Return Relations

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

Ex: Product(name, price, category, maker)

```
SELECT name
FROM   Product
WHERE  price > ALL(

    SELECT price
    FROM   Product
    WHERE  maker = 'Gizmo-Works')
```

Find products that are more expensive than all those produced by "Gizmo-Works"

```
SELECT p1.name
FROM   Product p1
WHERE  p1.maker = 'Gizmo-Works'
  AND EXISTS(
                 SELECT p2.name
           FROM  Product p2
          WHERE  p2.maker <> 'Gizmo-Works'
                   AND p1.name = p2.name)
```

<> means !=

Find 'copycat' products, i.e. products made by competitors with the same names as products made by "Gizmo-Works"

Note the scoping of the variables!

Example: Complex Correlated Query

Product(name, price, category, maker, year)

```
SELECT DISTINCT  x.name, x.maker
FROM             Product AS x
WHERE            x.price > ALL(
        SELECT y.price
        FROM   Product AS y
        WHERE  x.maker = y.maker
                         AND y.year < 1972)
```

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

Can be very powerful (also much harder to optimize)